1.

```
using System;

class Calculator
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter the first number:");
        double num1 = Convert.ToDouble(Console.ReadLine());

        Console.WriteLine("Enter the second number:");
        double num2 = Convert.ToDouble(Console.ReadLine());

        Console.WriteLine("Choose an operation: +, -, *, /");
        string operation = Console.ReadLine();

        double result = 0;

        switch (operation)
        {
            case "+":
                result = num1 + num2;
                break;
            case "-":
                result = num1 - num2;
                break;
            case "*":
                result = num1 * num2;
                break;
            case "/":
                if (num2 != 0)
                {
                    result = num1 / num2;
                }
                else
                {
                    Console.WriteLine("Error: Division by zero is not
allowed.");
                    return;
                }
                break;
            default:
                Console.WriteLine("Invalid operation.");
                return;
        }

        Console.WriteLine("Result: " + result);
    }
}
```

1(a)

```
using System;

class Calculator
{
    static void Main(string[] args)
    {
```

```
        Console.WriteLine("Enter numbers separated by spaces:");

        string input = Console.ReadLine();
        string[] inputArray = input.Split(' ');

        int[] numbers = new int[inputArray.Length];
        for (int i = 0; i < inputArray.Length; i++)
        {
            if (int.TryParse(inputArray[i], out int number))
            {
                numbers[i] = number;
            }
            else
            {
                Console.WriteLine($"'{inputArray[i]}' is not a valid
integer.");
                return;
            }
        }

        double average = CalculateAverage(numbers);
        Console.WriteLine("Average: " + average);
    }

    static double CalculateAverage(int[] numbers)
    {
        if (numbers == null || numbers.Length == 0)
        {
            return 0;
        }

        double sum = 0;
        foreach (int number in numbers)
        {
            sum += number;
        }

        return sum / numbers.Length;
    }
}
```

1(b)

Constructors are special methods in C# used to initialize objects when a class is instantiated.

Characteristics:

- ✓ Constructors are called automatically when an object of a class is created. They set up the initial state of the object, such as assigning values to fields.
- ✓ Constructors have the same name as the class and do not have a return type, not even void.
- ✓ Constructors can be overloaded. This means you can define multiple constructors with different parameters to create objects in different ways.
- ✓ Default Constructor: If no constructors are explicitly defined, C# provides a default constructor that initializes the object with default values (e.g., null for reference types, 0 for numeric types).

Difference Between Constructors and Other Methods:

- ✓ Constructors are automatically called when an object is instantiated, whereas other methods are explicitly called after an object has been created.
- ✓ The main purpose of constructors is to initialize the object, while other methods are used to define the behavior of the object.
- ✓ Constructors do not have a return type, unlike other methods that usually return a value or void.

Default Constructor and Overloaded Constructor

```
using System;

class Tracker
{
    public string Name { get; set; }
    public int Id { get; set; }

    // Default constructor
    public Tracker()
    {
        Name = "Unknown";
        Id = 0;
        Console.WriteLine("Default constructor called. Name: " + Name +
", Id: " + Id);
    }

    // Overloaded constructor
    public Tracker(string name, int id)
    {
        Name = name;
        Id = id;
        Console.WriteLine("Overloaded constructor called. Name: " + Name
+ ", Id: " + Id);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Using the default constructor
        Tracker tracker1 = new Tracker();

        // Using the overloaded constructor
        Tracker tracker2 = new Tracker("Item A", 1);
    }
}
```

1(c)

```
using System;

public class Employee
{
    public string Name { get; private set; }
    public int ID { get; private set; }
```

```
    public string Department { get; private set; }
    public decimal Salary { get; private set; }

    // Primary constructor
    public Employee(string name, int id)
    {
        Name = name;
        ID = id;
    }

    // Secondary constructor with optional parameters
    public Employee(string name, int id, string department = "Unknown", decimal
salary = 0.0m)
        : this(name, id)
    {
        Department = department;
        Salary = salary;
    }

    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, ID: {ID}, Department: {Department},
Salary: {Salary:C}");
    }
}

class Program
{
    static void Main()
    {
        // Creating an instance using the primary constructor
        Employee emp1 = new Employee("Charo Ishmael", 1234);
        emp1.DisplayInfo();

        // Creating an instance using the secondary constructor
        Employee emp2 = new Employee("Naomy Mwangi", 5678, "HR", 75000);
        emp2.DisplayInfo();
    }
}
```

2.

== Operator

- ✓ The == operator is used to check if two strings are **equal** in terms of their content.
- ✓ It compares the values of the two strings, meaning it checks if the sequences of characters in both strings are identical.
- ✓ The comparison is case-sensitive.
- ✓ If either string is null, the operator returns false unless both are null, in which case it returns true.

Equals()

- ✓ The Equals() method is used to compare the contents of two strings, with an option to specify whether the comparison should be case-sensitive or not.
- ✓ By default, it performs a case-sensitive comparison similar to the == operator, but it can be overloaded to perform case-insensitive comparisons.
- ✓ You can specify whether the comparison should be case-sensitive or case-insensitive using the overload Equals(string, StringComparison). For example,

string1.Equals(string2, StringComparison.OrdinalIgnoreCase) allows a case-insensitive comparison.

- ✓ If you call Equals() on a null string, it will throw a NullReferenceException. But if string1.Equals(string2) is used, it returns false when string1 is not null and string2 is null.

When to Use Each

- ✓ == used when we want to perform a straightforward comparison that is case-sensitive and doesn't require any specific handling of null values.
- ✓ **Equals()** used when we need more control over the comparison, such as performing a case-insensitive comparison or when comparing objects of different types that may not be strings.

2(a)

Console.WriteLine(str1 == str2);

- ✓ Prediction: TRUE, because str1 and str2 reference the same memory location.

Console.WriteLine(str1 == str3);

- ✓ Prediction: TRUE, because the contents of str1 and str3 are identical.

Console.WriteLine(str1.Equals(str3));

- ✓ Prediction: TRUE, because Equals() does not consider the memory reference but instead compares the sequence of characters in both strings hence str1 and str3 are the same.

3.

Common Language Runtime (CLR)

- ✓ The CLR manages the execution of .NET programs, providing important services such as memory management, exception handling, garbage collection, and security.
- ✓ It allows developers to write code in multiple languages like C#, VB.NET, F# and ensures that this code can run on any machine that has the .NET Framework.

Base Class Library (BCL)

- ✓ The BCL is a large collection of pre-built classes, functions, and types that developers can use to build their applications. It includes classes for basic data types, file I/O, networking and collections.

- ✓ The BCL provides a consistent, object-oriented API that developers can use to perform common tasks, so that they don't have to write code from scratch for these operations.

How They Work Together:

- ✓ When one writes and run a .NET application, the CLR is responsible for compiling and executing the code. It relies on the BCL to provide the necessary building blocks (classes, methods) that the code calls upon.
- ✓ The BCL, being a well-optimized set of libraries, ensures that the code you write is both efficient and reliable, while the CLR manages the execution environment, optimizing the performance of your application.

3(a)

```
using System;
using System.IO;

class LibraryManagement
{
    static void Main()
    {
        // Define the file path
        string filePath = "books.txt";

        // Create and write to the file
        CreateAndWriteToFile(filePath);

        // Read from the file
        ReadFromFile(filePath);
    }

    static void CreateAndWriteToFile(string filePath)
    {
        // List of books to write to the file
        string[] books = {
            "The Catcher in the Rye by J.D. Salinger",
            "To Kill a Mockingbird by Harper Lee",
            "1984 by George Orwell",
            "The Great Gatsby by F. Scott Fitzgerald",
            "Moby Dick by Herman Melville"
        };

        // Create or overwrite the file and write the list of books to it
        File.WriteAllLines(filePath, books);
        Console.WriteLine("File created and books written successfully.");
    }

    static void ReadFromFile(string filePath)
    {
        // Check if the file exists
        if (File.Exists(filePath))
        {
            // Read all lines from the file
            string[] books = File.ReadAllLines(filePath);

            // Display the contents of the file
            Console.WriteLine("Books in the file:");
            foreach (string book in books)
            {
```

```
                Console.WriteLine(book);
            }
        }
        else
        {
            Console.WriteLine("File not found.");
        }
    }
}
```

4.

Value Types

- Stored in **s**tack
- Memory allocation is directly on the stack.
- When you assign a value type to a variable, the actual value is stored, and any operation on the variable affects the value directly. Copying a value type variable creates a complete copy of the data.
- **Examples**:

    - ✓ int, char, bool, float, double
    - ✓ struct
    - ✓ enum

Reference Types

- Stored in the Heap (reference stored in the stack)
- A reference (or pointer) to the actual data is stored on the stack, while the data itself is stored on the heap.
- When you assign a reference type to a variable, you store the reference to the data, not the data itself. Modifying the variable through one reference affects the data seen by all references to that object.
- **Examples**:

    - ✓ class
    - ✓ Arrays
    - ✓ Delegates
    - ✓ Strings
    - ✓ Objects

Scenarios and Performance Considerations

Value Types:

- ✓ Since they are stored on the stack, access to value types is generally faster, but copying large value types (like large structs) can be costly.
- ✓ Use value types when you need a local copy of the data that won't be affected by changes elsewhere in the program.

Reference Types:

- ✓ Operations involving reference types can be slower due to heap allocation and garbage collection, but they are more efficient when passing around large objects or collections because only the reference is copied, not the data itself.
- ✓ Use reference types when you need to share data across different parts of your application or when you need to modify the data from different locations.

4(a)

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Value Type Example: int
        int a = 10;
        int b = a; // b is a copy of a

        Console.WriteLine("Value Type Example (int):");
        Console.WriteLine($"a: {a}, b: {b}");
        b = 20; // Changing b does not affect a
        Console.WriteLine($"After modifying b: a: {a}, b: {b}");
        Console.WriteLine();

        // Reference Type Example: string[]
        string[] arr1 = new string[] { "apple", "banana", "cherry" };
        string[] arr2 = arr1; // arr2 is a reference to arr1

        Console.WriteLine("Reference Type Example (string[]):");
        Console.WriteLine("arr1: " + string.Join(", ", arr1));
        Console.WriteLine("arr2: " + string.Join(", ", arr2));

        // Modifying arr2 affects arr1 because both refer to the same object
        arr2[1] = "blueberry";
        Console.WriteLine("After modifying arr2:");
        Console.WriteLine("arr1: " + string.Join(", ", arr1));
        Console.WriteLine("arr2: " + string.Join(", ", arr2));
        Console.WriteLine();

        // Comparing memory addresses using Object.ReferenceEquals
        Console.WriteLine("Comparing memory addresses:");
        Console.WriteLine($"Object.ReferenceEquals(arr1, arr2):
{Object.ReferenceEquals(arr1, arr2)}");

        // Creating a new array with the same content but different reference
        string[] arr3 = new string[] { "apple", "blueberry", "cherry" };
        Console.WriteLine($"Object.ReferenceEquals(arr1, arr3):
{Object.ReferenceEquals(arr1, arr3)}");

        // Value Type Comparison: Even though values are the same, they are
different in memory
        int x = 10;
        int y = 10;
        Console.WriteLine($"Object.ReferenceEquals(x, y):
{Object.ReferenceEquals(x, y)}"); // This will be false because value types are
stored separately

        // Reference Type Comparison with new reference
        string[] arr4 = new string[] { "apple", "blueberry", "cherry" };
```

```
        Console.WriteLine($"Object.ReferenceEquals(arr3, arr4):
{Object.ReferenceEquals(arr3, arr4)}"); // False, different objects
    }
}
```

5.

Encapsulation in C# is the binding of data (fields) and methods that operate on the data within a class, while restricting direct access to some of the class's components.

This is achieved by declaring fields as private or protected and providing public properties or methods to access and modify them.

- ✓ **Private Fields**: Fields are kept private to prevent unauthorized direct access.
- ✓ **Protected Fields**: These fields are accessible within the class where it is defined.
- ✓ **Public Properties/Methods**: Controlled access to these fields is provided controlled access to the protected or private fields.

Encapsulation ensures that an object's internal state is protected and can only be modified in controlled ways, enhancing security and integrity.

Encapsulation hides the internal implementation details of a class, which helps in preventing unauthorized access and reduces the complexity for users of the class.

5(a)

```
public class Person
{
    // Private fields
    private string name;
    private int age;

    // Public property for 'name' with basic encapsulation
    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
            {
                name = value;
            }
            else
            {
                throw new ArgumentException("Name cannot be null or empty.");
            }
        }
    }

    // Public property for 'age' with validation
    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
            {
                age = value;
```

```
            }
            else
            {
                throw new ArgumentOutOfRangeException("Age cannot be
negative.");
            }
        }
    }
}
```

6.

A **single-dimensional array** is a basic array type that stores elements in a linear, contiguous block of memory. Each element in this array can be accessed via a single index.

A **jagged array**, is an array of arrays. Each element of the jagged array can be an array of a different size or dimension. It is more flexible as it allows rows of different lengths.

Use Cases:

Single-Dimensional Array:

- You want to store and process a simple list of values, such as the ages of employees in a company.
- **Example:** `int[] ages = new int[] { 25, 30, 45, 60 };`

Jagged Array:

- You need to represent a collection of data where the number of elements in each collection varies, such as representing a list of employees in different departments, where each department has a different number of employees.
- **Example:**

```
int[][] departments = new int[3][];
departments[0] = new int[] { 1, 2 };   // Department 1 has 2
employees
departments[1] = new int[] { 3, 4, 5 }; // Department 2 has 3
employees
departments[2] = new int[] { 6 };      // Department 3 has 1
employee
```

6 (a)

method

```
public int SumOfElements(int?[][] array)
{
    int sum = 0;

    foreach (var row in array)
    {
        if (row != null) // Check if the row is not null
        {
            foreach (var element in row)
            {
                if (element.HasValue) // Check if the element is not null
                {
                    sum += element.Value;
```

```
                }
            }
        }
    }

    return sum;
}
```

6 (b)

enum

```
public enum Color
{
    Red,
    Green,
    Blue,
}

public class Shape
{
    public class Circle
    {
        public Color CircleColor { get; set; }

        public Circle(Color color)
        {
            CircleColor = color;
        }

        public void DisplayColor()
        {
            Console.WriteLine($"The circle color is: {CircleColor}");
        }
    }
}
```

7.

How Exceptions are Handled:

try Block:

- Code that may throw an exception is placed inside the try block.
- If an exception occurs, the runtime jumps to the catch block that matches the type of exception thrown.

catch Block:

- The catch block is where you handle the exception. You can have multiple catch blocks to handle different types of exceptions.

2. **finally Block:**
    - The finally block is optional and is used for cleanup code that must execute whether an exception is thrown or not. Examples include closing files or releasing resources.

Best Practices:

1. **Specific Exception Handling**
    - Catch specific exceptions rather than using a general catch (Exception ex) block. This makes your code more predictable and easier to debug.
2. **Use finally for Cleanup**
    - Use the finally block to ensure that resources like file handles or database connections are properly closed, even if an exception occurs.
3. **Minimal Use of Exceptions for Flow Control**
    - Avoid using exceptions to control the flow of the program (e.g., as part of normal logic). Exceptions should only be used for exceptional situations.
4. **Re-throwing Exceptions:**
    - If you catch an exception but can't handle it, consider re-throwing it to be handled by a higher-level component.

Potential Pitfalls:

1. **Swallowing Exceptions:**
    - Avoid catching exceptions without handling them. It can hide errors and make debugging difficult.
2. **Overusing General Catch Blocks:**
    - Catching the base Exception type without narrowing down the specific exceptions can lead to unintended behaviors and make diagnosing issues harder.
3. **Performance Overhead:**
    - Throwing and catching exceptions can be expensive in terms of performance, so avoid relying on them for routine operations.

7 (a)

```
using System;

class ListManagement
{
    static void Main()
    {
        int[] numbers = { 1, 2, 3, 4, 5 };

        try
        {
            Console.WriteLine("Enter an index to access an element from the
array:");
            string userInput = Console.ReadLine();

            try
            {
                // Trying to parse the user's input into an integer
```

```
                int index = int.Parse(userInput);

                try
                {
                    // Attempting to access an element in the array
                    int value = numbers[index];
                    Console.WriteLine($"Value at index {index}: {value}");
                }
                catch (IndexOutOfRangeException ex)
                {
                    // Handling array index out-of-bounds exception
                    Console.WriteLine("Error: Index out of bounds.");
                    Console.WriteLine("Details: " + ex.Message);
                }
            }
            catch (FormatException ex)
            {
                // Handling invalid number format
                Console.WriteLine("Error: Input is not a valid number.");
                Console.WriteLine("Details: " + ex.Message);
            }
        }
        catch (Exception ex)
        {
            // General exception handler for unexpected errors
            Console.WriteLine("An unexpected error occurred.");
            Console.WriteLine("Details: " + ex.Message);
        }
        finally
        {
            // Cleanup code
            Console.WriteLine("Program execution completed.");
        }
    }
}
```

8.

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter an integer: ");
        int number = int.Parse(Console.ReadLine());

        if (number == 0)
        {
            Console.WriteLine("The number is zero.");
        }
        else
        {
            if (number > 0)
            {
                Console.WriteLine("The number is positive.");
            }
            else
            {
                Console.WriteLine("The number is negative.");
            }

            if (number % 2 == 0)
```

```
            {
                Console.WriteLine("The number is even.");
            }
            else
            {
                Console.WriteLine("The number is odd.");
            }
        }
    }
}
```

8 (a)

While Loop

The while loop continuously executes a block of code as long as a specified condition is true. The condition is evaluated before the execution of the loop's body. If the condition is initially false, the loop body will not execute even once.

Syntax:

while (condition)

{

// Code to be executed

}

Example:

```
int count = 0;

while (count < 5)

{

    Console.WriteLine("Count is: " + count);

    count++;

}
```

Scenario: While loop is suitable when the number of iterations is not known beforehand and the loop should continue until a certain condition is met. For example, reading user input until a valid value is entered.

Do-While Loop

The do-while loop is similar to the while loop, but the key difference is that the condition is evaluated after the loop body has executed. This guarantees that the loop body will execute at least once, regardless of the condition.

Syntax:

do

{

// Code to be executed

} while (condition);

Example:

```
int number;

do

{

    Console.Write("Enter a positive number: ");

    number = int.Parse(Console.ReadLine());

} while (number <= 0);
```

Scenario: The do-while loop is useful when you need the code inside the loop to execute at least once before checking the condition, such as when prompting a user for input where you want to ensure that the input is handled at least once.

For Loop

The for loop is often used when the number of iterations is known beforehand. It consists of three main parts: initialization, condition, and iteration expression. The initialization happens once at the beginning, the condition is evaluated before each loop iteration, and the iteration expression is executed after each iteration of the loop body.

Syntax:

for (initialization; condition; iteration)

{

  // Code to be executed

}

Example:

```
for (int i = 0; i < 5; i++)

{

    Console.WriteLine("Iteration: " + i);

}
```

Scenario: The for loop is ideal when you know the exact number of iterations in advance, such as when iterating over arrays, lists, or when counting from a starting value to an ending value.

8 (b)

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter an odd number: ");
        int number = int.Parse(Console.ReadLine());

        // Check if the number is odd
        if (number % 2 == 0)
        {
```

```
            Console.WriteLine("The number is even. Please enter an odd
number.");
        }
        else
        {
            long factorial = 1;

            // Calculate factorial using a for loop
            for (int i = 1; i <= number; i++)
            {
                factorial *= i;
            }

            Console.WriteLine($"The factorial of {number} is: {factorial}");
        }
    }
}
```

8 (c)

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter the number of rows for the triangle: ");
        int rows = int.Parse(Console.ReadLine());

        // Right-angled triangle pattern
        Console.WriteLine("Right-Angled Triangle:");
        for (int i = 1; i <= rows; i++)
        {
            for (int j = 1; j <= i; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }

        // Inverted right-angled triangle pattern
        Console.WriteLine("\nInverted Right-Angled Triangle:");
        for (int i = rows; i >= 1; i--)
        {
            for (int j = 1; j <= i; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    }
}
```

9.

Role of Threads in C#

Threads allow for concurrent execution, enabling a program to perform multiple tasks simultaneously. Threads can improve performance, responsiveness, and resource utilization, especially in applications that involve I/O-bound or CPU-bound operations.

Threads provide a way to create, manage, and control multiple paths of execution within a single application. By running tasks concurrently on different threads, programs can handle multiple operations without waiting for one to complete before starting another.

Thread Class vs. Task Class

1. **Thread Class:**
    - The Thread class represents a managed thread in the .NET environment. It allows low-level control over the thread, including starting, stopping, and pausing the thread.
    - Threads are ideal when you need direct control over the thread's lifecycle, such as managing priorities, setting apartment states, or when performing long-running tasks that do not require a return value.

When to Use: Use Thread when you need more control over the execution, such as when managing multiple threads with specific behavior (e.g., long-running background tasks or complex synchronization requirements).

2. **Task Class:**
    - The Task class, part of the System.Threading.Tasks namespace, is higher-level and focuses on asynchronous programming and parallelism. Unlike Thread, the Task class provides built-in support for managing the lifecycle of operations, including continuation tasks, exceptions, and cancellation.
    - Tasks are preferable for short-lived operations, I/O-bound work, or when working with async/await for non-blocking operation

When to Use: Use Task for high-level parallelism or when performing asynchronous operations, such as network calls, file I/O, or any other scenario where non-blocking behavior is beneficial. The Task class is ideal when you want to use the async and await keywords to simplify asynchronous code.

Key Differences:

- **Control:** The Thread class provides low-level control over the thread's lifecycle, whereas the Task class abstracts much of the complexity and focuses on parallelism and asynchronous programming.
- **Performance:** Tasks are generally more lightweight and efficient for short-lived operations, while threads are more appropriate for long-running, dedicated background work.
- **Async/Await:** The Task class integrates seamlessly with the async/await pattern, making it a better choice for non-blocking, asynchronous code.

9 (a)

```
using System;
using System.Threading;

class Program
{
    private static object _lock = new object();
    private static int _sharedCounter = 0;
```

```
    static void IncrementCounter()
    {
        // Thread-safe increment
        for (int i = 0; i < 5; i++)
        {
            lock (_lock)  // Ensuring thread safety
            {
                int temp = _sharedCounter;
                Thread.Sleep(100); // Simulate work
                _sharedCounter = temp + 1;
                Console.WriteLine($"Thread incremented counter to:
{_sharedCounter}");
            }
        }
    }

    static void Main()
    {
        // Create and start a new thread
        Thread newThread = new Thread(IncrementCounter);
        newThread.Start();

        // Perform operations on the main thread
        for (int i = 0; i < 5; i++)
        {
            lock (_lock)  // Ensuring thread safety
            {
                int temp = _sharedCounter;
                Thread.Sleep(100); // Simulate work
                _sharedCounter = temp + 1;
                Console.WriteLine($"Main thread incremented counter to:
{_sharedCounter}");
            }
        }

        // Wait for the new thread to complete
        newThread.Join();

        Console.WriteLine("Both threads have completed.");
    }
}
```

10.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Text.Json;

class Program
{
    private static readonly string apiKey = "AIzaSyDaGmWKa4JsXZ-
HjGw7ISLn_3namBGewQe";
    private static readonly string apiUrl = "https://newsapi.org/v2/top-
headlines?country=us&apiKey=" + apiKey;

    static async Task Main()
    {
        // Create HttpClient instance
        using HttpClient client = new HttpClient();

        try
        {
```

```
            // Make a GET request to the API
            HttpResponseMessage response = await client.GetAsync(apiUrl);
            response.EnsureSuccessStatusCode();

            // Read and parse the JSON response
            string responseBody = await response.Content.ReadAsStringAsync();
            var newsResponse =
JsonSerializer.Deserialize<NewsResponse>(responseBody);

            // Display article titles and summaries
            Console.WriteLine("Top News Headlines:");
            foreach (var article in newsResponse.Articles)
            {
                Console.WriteLine($"Title: {article.Title}");
                Console.WriteLine($"Summary: {article.Description}");
                Console.WriteLine();
            }
        }
        catch (HttpRequestException e)
        {
            Console.WriteLine($"Request error: {e.Message}");
        }
    }
}

// Define classes for JSON deserialization
public class NewsResponse
{
    public Article[] Articles { get; set; }
}

public class Article
{
    public string Title { get; set; }
    public string Description { get; set; }
}
```

10 (a)

```
using System;

using System.IO;

using System.Linq;


class Program

{

    static void Main()

    {

        string inputFilePath = "input.txt";  // Path to the input file

        string outputFilePath = "output.txt"; // Path to the output file


        // Keywords to filter lines

        string[] keywords = { "important", "urgent", "review" };
```

```
        int minLength = 20; // Minimum length for lines to be included


        try
        {
            // Open the input file for reading
            using (StreamReader reader = new StreamReader(inputFilePath))
            {
                // Open the output file for writing
                using (StreamWriter writer = new
StreamWriter(outputFilePath))
                {
                    string line;


                    // Read each line from the input file
                    while ((line = reader.ReadLine()) != null)
                    {
                        // Check if the line contains any of the keywords
or is longer than the minimum length
                        if (keywords.Any(keyword => line.IndexOf(keyword,
StringComparison.OrdinalIgnoreCase) >= 0) || line.Length >= minLength)
                        {
                            // Write the filtered line to the output file
                            writer.WriteLine(line);
                        }
                    }
                }
            }


            Console.WriteLine("Filtering complete. Check the output
file.");
        }
        catch (IOException e)
        {
            Console.WriteLine($"An error occurred: {e.Message}");
        }
    }
```

```
}
```

11.

Packages in C#

In C#, packages (also known as libraries or NuGet packages) are collections of pre-written code that can be reused in different projects. They simplify development by providing ready-to-use functionality, saving time and effort in writing and maintaining code. Packages can include libraries, tools, or frameworks that extend the capabilities of your application.

Key Purposes:

1. **Code Reusability:** Packages allow you to reuse code across multiple projects without reinventing the wheel.
2. **Simplified Development:** Packages provide well-tested solutions for common tasks, reducing the complexity of your development process.
3. **Consistency:** Using packages helps ensure that the same version of a library is used consistently across different projects or parts of a project.
4. **Dependency Management:** Packages manage dependencies and versioning, making it easier to update libraries and avoid conflicts.

Installing and Using a NuGet Package

1. **Install a NuGet Package:**
    - **Via Visual Studio:**
        1. Right-click on your project in Solution Explorer.
        2. Select Manage NuGet Packages.
        3. Search for the desired package (e.g., Newtonsoft.Json for JSON handling).
        4. Click Install to add the package to your project.
    - **Via .NET CLI:**

        ```
        dotnet add package <PackageName>
        ```

        For example, to install Newtonsoft.Json

        ```
        dotnet add package Newtonsoft.Json
        ```
2. **Use the Package:**
    - After installation, you can use the package in your code by adding the appropriate using directive. e.g `using Newtonsoft.Json;`

11 (a)

```
using System;
```

```
using System.Collections.Generic;
using System.Xml;
using Newtonsoft.Json;

class Program
{
    public class Address
    {
        public string Street { get; set; }
        public string City { get; set; }
        public string ZipCode { get; set; }
    }

    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public Address Address { get; set; }
        public List<string> PhoneNumbers { get; set; }
    }

    static void Main()
    {
        // Create a Person object
        Person person = new Person
        {
            Name = "John Doe",
            Age = 30,
            Address = new Address
            {
                Street = "123 Elm Street",
                City = "Somewhere",
                ZipCode = "12345"
            },
            PhoneNumbers = new List<string> { "555-1234", "555-5678" }
        };

        // Serialize the Person object to JSON
        string jsonString = JsonConvert.SerializeObject(person,
Formatting.Indented);
        Console.WriteLine("Serialized JSON:");
        Console.WriteLine(jsonString);

        // Deserialize the JSON back to a Person object
        Person deserializedPerson =
JsonConvert.DeserializeObject<Person>(jsonString);
        Console.WriteLine("\nDeserialized Person:");
        Console.WriteLine($"Name: {deserializedPerson.Name}");
        Console.WriteLine($"Age: {deserializedPerson.Age}");
        Console.WriteLine($"Street: {deserializedPerson.Address.Street}");
        Console.WriteLine($"City: {deserializedPerson.Address.City}");
        Console.WriteLine($"ZipCode: {deserializedPerson.Address.ZipCode}");
        Console.WriteLine($"Phone Numbers: {string.Join(", ",
deserializedPerson.PhoneNumbers)}");
    }
}
```

12.

1. List<T>

- List<T> is a dynamic array that allows adding, accessing, and modifying elements by index. It can grow and shrink as needed, and it maintains the order in which elements are added.
- **Operations**:
  - ✓ **Add**: Adds an element to the end of the list.
  - ✓ **Insert**: Inserts an element at a specified index.
  - ✓ **Remove**: Removes the first occurrence of a specific element or removes an element by index.
  - ✓ **Indexing**: Access elements by index.
- **Use Cases**:
  - ✓ When you need to maintain a collection of items where random access and modification by index are important.
  - ✓ Suitable for situations where the number of elements can vary and operations like searching and iterating are common.

  Example: Storing a list of student names where you might need to access, modify, or search for specific students by index.

2. Queue<T>

- Queue<T> is a first-in, first-out (FIFO) collection. Elements are added to the end (enqueued) and removed from the front (dequeued). It is useful for processing elements in the order they were added.
- **Operations**:
  - ✓ **Enqueue**: Adds an element to the end of the queue.
  - ✓ **Dequeue**: Removes and returns the element at the front of the queue.
  - ✓ **Peek**: Returns the element at the front of the queue without removing it.
- **Use Cases**:
  - ✓ When you need to process elements in the order they arrive.
  - ✓ Useful in scenarios like task scheduling, printer job management, or breadth-first search in graph algorithms.

  Example: A task processing system where tasks are processed in the order they are added.

3. Stack<T>

- Stack<T> is a last-in, first-out (LIFO) collection. Elements are added and removed from the top of the stack. This structure is useful for scenarios where the most recent element needs to be processed first.
- **Operations**:

- ✓ **Push**: Adds an element to the top of the stack.
- ✓ **Pop**: Removes and returns the element at the top of the stack.
- ✓ **Peek**: Returns the element at the top of the stack without removing it.

- **Use Cases**:
  - ✓ When you need to process elements in reverse order, or "undo" operations.
  - ✓ Suitable for scenarios like expression evaluation, function call management (recursive algorithms), or implementing backtracking.

  Example: Implementing an undo functionality in a text editor where the last action needs to be reversed first.

When to Use Each Data Structure

- **Use List<T>** when you need:
  - ✓ Random access by index.
  - ✓ Flexibility in inserting/removing items from any position.
  - ✓ A dynamic array that can grow or shrink.
- **Use Queue<T>** when you need:
  - ✓ FIFO behavior (first come, first served).
  - ✓ To manage tasks, requests, or processes in the order they arrive.
  - ✓ To traverse levels of a tree or graph in order.
- **Use Stack<T>** when you need:
  - ✓ LIFO behavior (last in, first out).
  - ✓ To implement "undo" features, function call stacks, or nested structures.
  - ✓ To reverse processes, such as parsing expressions or backtracking algorithms.

12 (a)

```
using System;
using System.Collections.Generic;

class Program
{
    // Enum to define customer types
    enum CustomerType
    {
        Regular,
        VIP
    }

    // Class to represent a customer
    class Customer
    {
        public string Name { get; set; }
        public CustomerType Type { get; set; }
```

```
        public Customer(string name, CustomerType type)
        {
            Name = name;
            Type = type;
        }

        public override string ToString()
        {
            return $"{Name} ({Type})";
        }
    }

    // Custom comparer to prioritize VIPs over Regular customers
    class CustomerComparer : IComparer<Customer>
    {
        public int Compare(Customer x, Customer y)
        {
            if (x.Type == y.Type)
                return 0;

            // VIPs should come first
            return x.Type == CustomerType.VIP ? -1 : 1;
        }
    }

    static void Main(string[] args)
    {
        // Priority queue where VIPs have higher priority than regular
customers
        PriorityQueue<Customer, Customer> bankQueue = new
PriorityQueue<Customer, Customer>(new CustomerComparer());

        // Add customers to the queue
        bankQueue.Enqueue(new Customer("Alice", CustomerType.Regular), new
Customer("Alice", CustomerType.Regular));
        bankQueue.Enqueue(new Customer("Bob", CustomerType.VIP), new
Customer("Bob", CustomerType.VIP));
        bankQueue.Enqueue(new Customer("Charlie", CustomerType.Regular), new
Customer("Charlie", CustomerType.Regular));
        bankQueue.Enqueue(new Customer("Diana", CustomerType.VIP), new
Customer("Diana", CustomerType.VIP));
        bankQueue.Enqueue(new Customer("Eve", CustomerType.Regular), new
Customer("Eve", CustomerType.Regular));

        // Process the queue
        Console.WriteLine("Processing bank queue:");
        while (bankQueue.Count > 0)
        {
            Customer nextCustomer = bankQueue.Dequeue();
            Console.WriteLine($"Serving: {nextCustomer}");
        }
    }
}
```

NB: The above implementation has used priorityQueue<T>.

13.

Inheritance in C#

Inheritance is one of the core principles of Object-Oriented Programming (OOP) in C#. It allows a class (called the derived or child class) to inherit properties, methods, and behaviors

from another class (called the base or parent class). This promotes code reuse and a hierarchical class structure.

Key Concepts of Inheritance

1. **Base Class**: The class from which properties and methods are inherited.
2. **Derived Class**: The class that inherits from the base class.
3. **Access Modifiers**: Define the accessibility of members (fields, properties, methods) in classes and their derived classes.
    - ✓ public: Members are accessible from any class.
    - ✓ protected: Members are accessible within the base class and its derived classes.
    - ✓ private: Members are accessible only within the base class.
    - ✓ internal: Members are accessible within the same assembly.
    - ✓ protected internal: Members are accessible within the same assembly or derived classes.

13 (a)

```
using System;

class Animal
{
    // Protected member, accessible in derived classes
    protected string name;

    public Animal(string name)
    {
        this.name = name;
    }

    // Virtual method that can be overridden in derived classes
    public virtual void Speak()
    {
        Console.WriteLine($"{name} makes a sound.");
    }
}

class Dog : Animal
{
    public Dog(string name) : base(name) { }

    // Override the Speak method to provide specific behavior for Dog
    public override void Speak()
    {
        Console.WriteLine($"{name} barks.");
    }
}

class Cat : Animal
{
    public Cat(string name) : base(name) { }

    // Override the Speak method to provide specific behavior for Cat
```

```
    public override void Speak()
    {
        Console.WriteLine($"{name} meows.");
    }
}

class Bird : Animal
{
    public Bird(string name) : base(name) { }

    // Override the Speak method to provide specific behavior for Bird
    public override void Speak()
    {
        Console.WriteLine($"{name} chirps.");
    }
}

class Program
{
    static void Main()
    {
        // Polymorphism: Treat derived classes as their base class
        Animal myDog = new Dog("Rex");
        Animal myCat = new Cat("Whiskers");
        Animal myBird = new Bird("Tweety");

        // Demonstrate polymorphism
        myDog.Speak();  // Rex barks.
        myCat.Speak();  // Whiskers meows.
        myBird.Speak(); // Tweety chirps.

        // Array of Animals demonstrating polymorphism
        Animal[] animals = { myDog, myCat, myBird };
        foreach (Animal animal in animals)
        {
            animal.Speak(); // Each animal speaks according to its type
        }
    }
}
```

14.

Polymorphism in C#

Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP) in C#, along with inheritance, encapsulation, and abstraction. Polymorphism allows objects of different types to be treated as objects of a common base type, enabling the same operation to behave differently on different objects. This promotes flexibility and scalability in code.

Polymorphism can be achieved in C# in two main ways:

1. **Compile-Time (Static) Polymorphism**: Achieved using method overloading and operator overloading.
2. **Run-Time (Dynamic) Polymorphism**: Achieved through method overriding using inheritance.

1. Compile-Time Polymorphism

This is resolved during compilation. The method to be invoked is determined based on the method signature, i.e., the method name, parameters, and their types.

- **Method Overloading**: Multiple methods with the same name but different signatures (different number or types of parameters).

```
class Calculator
{
    public int Add(int a, int b)
   {
       return a + b;
   }

   public double Add(double a, double b)
   {
        return a + b;
    }
}

class Program
{
    static void Main()
   {
       Calculator calc = new Calculator();
        Console.WriteLine(calc.Add(2, 3));      // Calls Add(int, int)
        Console.WriteLine(calc.Add(2.5, 3.5));  // Calls Add(double,
double)
    }
}
```

2. Run-Time Polymorphism

This is resolved during runtime. The actual method to be invoked is determined based on the runtime type of the object. It is achieved using inheritance and method overriding.

- **Method Overriding**: The base class provides a method, and the derived class overrides that method to provide its own implementation. This is done using the virtual keyword in the base class and the override keyword in the derived class.

```
using System;

class Animal
{
    // Virtual method can be overridden in derived classes
    public virtual void Speak()
    {
        Console.WriteLine("The animal makes a sound.");
    }
}

class Dog : Animal
{
    // Override the base class method
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}

class Cat : Animal
{
    // Override the base class method
    public override void Speak()
```

```
        {
            Console.WriteLine("The cat meows.");
        }
    }

    class Bird : Animal
    {
        // Override the base class method
        public override void Speak()
        {
            Console.WriteLine("The bird chirps.");
        }
    }

    class Program
    {
        static void Main()
        {
            // Polymorphism: base class reference points to derived class
    objects
            Animal myDog = new Dog();
            Animal myCat = new Cat();
            Animal myBird = new Bird();

            // Polymorphic behavior: calling Speak() on different objects
           myDog.Speak();  // Output: The dog barks.
           myCat.Speak();  // Output: The cat meows.
           myBird.Speak(); // Output: The bird chirps.

            // Array of Animal references
            Animal[] animals = { myDog, myCat, myBird };
            foreach (Animal animal in animals)
            {
                animal.Speak(); // Each derived class's overridden Speak
    method is called
            }
        }
    }
```

14 (a)

```
using System;

// Define the interface IDrive
interface IDrive
{
    void Drive();  // Method to be implemented by classes
}

// Base class Vehicle
abstract class Vehicle : IDrive
{
    public string Make { get; set; }
    public string Model { get; set; }

    public Vehicle(string make, string model)
    {
        Make = make;
        Model = model;
    }

    // Abstract method to force derived classes to implement it
    public abstract void Drive();
```

```
    // Common method for all vehicles
    public void ShowDetails()
    {
        Console.WriteLine($"Vehicle: {Make} {Model}");
    }
}

// Derived class Car
class Car : Vehicle
{
    public int NumberOfDoors { get; set; }

    public Car(string make, string model, int numberOfDoors) : base(make,
model)
    {
        NumberOfDoors = numberOfDoors;
    }

    // Implement the Drive method specific to Car
    public override void Drive()
    {
        Console.WriteLine($"Driving a {Make} {Model} car with {NumberOfDoors}
doors.");
    }
}

// Derived class Bike
class Bike : Vehicle
{
    public bool HasPedals { get; set; }

    public Bike(string make, string model, bool hasPedals) : base(make, model)
    {
        HasPedals = hasPedals;
    }

    // Implement the Drive method specific to Bike
    public override void Drive()
    {
        string pedalStatus = HasPedals ? "with pedals" : "without pedals";
        Console.WriteLine($"Riding a {Make} {Model} bike {pedalStatus}.");
    }
}

class Program
{
    static void Main()
    {
        // Create instances of Car and Bike
        Vehicle myCar = new Car("Toyota", "Corolla", 4);
        Vehicle myBike = new Bike("Yamaha", "MT-15", false);

        // Demonstrate polymorphism
        myCar.Drive();  // Driving a Toyota Corolla car with 4 doors.
        myBike.Drive(); // Riding a Yamaha MT-15 bike without pedals.

        // Show details of each vehicle
        myCar.ShowDetails(); // Vehicle: Toyota Corolla
        myBike.ShowDetails(); // Vehicle: Yamaha MT-15

        // Array of vehicles demonstrating polymorphism
        Vehicle[] vehicles = { myCar, myBike };
        foreach (Vehicle vehicle in vehicles)
```

```
        {
            vehicle.Drive(); // Polymorphic behavior based on the actual object
type
        }
    }
}
```

15.

Abstraction iis the concept of hiding the internal implementation details of an object and exposing only the necessary functionalities. The goal of abstraction is to simplify complex systems by breaking them down into more manageable pieces, focusing on what an object does rather than how it does it.

Abstraction can be implemented in C# using:

1. **Abstract Classes**
2. **Interfaces**

1. Abstract Classes

An abstract class is a class that cannot be instantiated on its own. It serves as a base class for other classes. Abstract classes can have both abstract methods (without implementation) and regular methods (with implementation). Derived classes are required to implement the abstract methods defined in the abstract class.

- **When to Use:** Abstract classes are useful when you want to provide a common base class that includes some shared implementation, but you also want derived classes to provide specific behaviors for some methods.

2. Interfaces.

An interface in C# is a contract that defines a set of methods and properties that a class must implement. Unlike abstract classes, interfaces cannot have any implementation. A class that implements an interface agrees to provide implementations for all the members defined in the interfaces.

- **When to Use:** Interfaces are ideal when you want to define a contract that multiple classes should adhere to, without worrying about how they implement the details. Since a class can implement multiple interfaces, this approach supports multiple inheritance in C#.

How Abstraction Simplifies Code and Enhances Maintainability

1. **Code Reusability:** Abstraction allows common functionality to be written in a base class or defined in an interface. This means that the same logic can be reused by multiple derived classes, reducing duplication and enhancing maintainability.
    - **Example:** An abstract Shape class with a method CalculateArea() can be implemented by derived classes like Circle, Rectangle, and Triangle. This allows new shapes to be added easily by simply extending the base class.

2. **Reduced Complexity:** By hiding the implementation details and exposing only essential behaviors, abstraction reduces the complexity of the code that interacts with the objects. The consumer of a class does not need to know how methods are implemented, only that they exist.
    - **Example:** A PaymentProcessor interface can abstract different payment methods such as CreditCardPayment, PayPalPayment, and BankTransferPayment, making it easy to switch between different payment types without altering the core logic of the application.
3. **Enhancing Flexibility:** Abstraction provides flexibility by allowing different implementations to exist under a common interface or base class. This makes the system adaptable to change. You can add new implementations without altering existing code, adhering to the Open/Closed Principle of SOLID design.
    - **Example:** A Logger interface can have different implementations such as FileLogger, DatabaseLogger, and ConsoleLogger. The rest of the application only needs to work with the Logger interface, making it easy to change or extend logging mechanisms.
4. **Improving Testability:** Abstraction simplifies unit testing by enabling mock implementations. By relying on interfaces, you can create mock objects for testing, isolating specific parts of the system and improving the overall testability.
    - **Example:** If a UserService class depends on a IEmailSender interface, you can mock the IEmailSender during unit tests, avoiding the need to send real emails during testing.

Scenarios Where Abstraction Enhances Maintainability

1. **Payment Processing System:** Different payment gateways (e.g., Stripe, PayPal) can be abstracted using an interface like IPaymentProcessor. This way, adding or switching to a new gateway only requires implementing the interface without modifying the rest of the payment system logic.
2. **Data Access Layer:** A repository pattern can be abstracted using interfaces. For example, an IRepository<T> interface can define basic CRUD operations. This allows switching between different data storage mechanisms (e.g., SQL, NoSQL) by simply implementing the interface.
3. **User Authentication System:** Different authentication mechanisms (e.g., OAuth, JWT, LDAP) can be abstracted behind an IAuthenticationService interface. This makes it easy to swap or extend authentication methods without modifying the application's core logic.

15 (a)

Abstract class Shape.

```
using System;

abstract class Shape
{
```

```
    // Common property for all shapes
    public string Color { get; set; }

    // Constructor to set the color of the shape
    public Shape(string color)
    {
        Color = color;
    }

    // Abstract method that must be implemented by derived classes
    public abstract void Draw();

    // Common method for all shapes
    public void Describe()
    {
        Console.WriteLine($"This is a {Color} shape.");
    }
}
```

Derived class Circle.

```
using System.Drawing;

class Circle : Shape
{
    public double Radius { get; set; }

    // Constructor that sets the color and radius of the circle
    public Circle(string color, double radius) : base(color)
    {
        Radius = radius;
    }

    // Implementation of the abstract Draw method
    public override void Draw()
    {
        Console.WriteLine($"Drawing a {Color} circle with a radius of
{Radius}.");
    }

    // Additional method specific to Circle
    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

Derived Class Square.

```
using System.Drawing;

class Square : Shape
{
    public double SideLength { get; set; }

    // Constructor that sets the color and side length of the square
    public Square(string color, double sideLength) : base(color)
    {
        SideLength = sideLength;
    }

    // Implementation of the abstract Draw method
```

```
    public override void Draw()
    {
        Console.WriteLine($"Drawing a {Color} square with a side length of
{SideLength}.");
    }

    // Additional method specific to Square
    public double CalculatePerimeter()
    {
        return 4 * SideLength;
    }
}
```

16. The output will be:

1

2

3

4

5

16 (a)

The output will be: *True*

16 (b)

The output will be: *False*

16 (c)

The output will be: *15*

17.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    public static (int? smallest, int? largest)
FindSmallestAndLargest(List<int> numbers)
    {
        if (numbers == null || numbers.Count == 0)
        {
            // Return null for both smallest and largest when the list is empty
            return (null, null);
        }

        int smallest = numbers.Min();
        int largest = numbers.Max();

        return (smallest, largest);
    }
```

```
    static void Main()
    {
        List<int> numbers = new List<int> { 3, 7, 2, 9, 4 };

        var result = FindSmallestAndLargest(numbers);
        if (result.smallest.HasValue && result.largest.HasValue)
        {
            Console.WriteLine($"Smallest: {result.smallest}, Largest:
{result.largest}");
        }
        else
        {
            Console.WriteLine("The list is empty.");
        }
    }
}
```

17 (a)

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        HashSet<int> uniqueNumbers = new HashSet<int>();
        int sum = 0;

        while (true)
        {
            Console.Write("Enter an integer (negative number to stop): ");
            int number = int.Parse(Console.ReadLine());

            if (number < 0)
            {
                break;
            }

            if (uniqueNumbers.Add(number)) // Adds to the set and checks if it
was added successfully (i.e., it's unique)
            {
                sum += number;
            }
        }

        Console.WriteLine($"Sum of unique entered integers: {sum}");
    }
}
```

17 (b)

```
using System;

class Program
{
    enum DaysOfWeek
    {
        Sunday,
        Monday,
        Tuesday,
```

```
        Wednesday,
        Thursday,
        Friday,
        Saturday
    }

    static void Main()
    {
        DaysOfWeek today = DaysOfWeek.Saturday;

        Console.WriteLine($"Today is: {today}");

        // Check if today is a weekend day
        if (today == DaysOfWeek.Saturday || today == DaysOfWeek.Sunday)
        {
            Console.WriteLine("It's a weekend!");
        }
        else
        {
            Console.WriteLine("It's a weekday.");
        }

        // Perform different operations based on the day
        switch (today)
        {
            case DaysOfWeek.Monday:
                Console.WriteLine("Start of the work week.");
                break;
            case DaysOfWeek.Wednesday:
                Console.WriteLine("Middle of the week.");
                break;
            case DaysOfWeek.Friday:
                Console.WriteLine("Almost the weekend!");
                break;
            default:
                Console.WriteLine("Just another day.");
                break;
        }
    }
}
```

17 (c)

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();

        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);
        string reversedString = new string(charArray);

        Console.WriteLine($"Reversed string: {reversedString}");
    }
}
```

17 (d)

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Dictionary to store student grades with different key-value types
        Dictionary<object, object> studentGrades = new Dictionary<object,
object>();

        // Adding entries with different types of keys and values
        studentGrades.Add("John", 85);                              // String key,
int value
        studentGrades.Add(1001, "A+");                              // int key,
string value
        studentGrades.Add(3.14159, new List<int> { 90, 92, 88 }); // double
key, List<int> value
        studentGrades.Add(true, 'B');                               // bool key,
char value
        studentGrades.Add(new DateTime(2024, 5, 1), 75.5);       // DateTime
key, double value

        // Display all entries in the dictionary
        Console.WriteLine("Available Entries in the Dictionary:");
        foreach (var entry in studentGrades)
        {
            Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
        }

        // Prompt the user to enter a key to retrieve a value
        Console.WriteLine("\nEnter a key to retrieve the corresponding
grade:");
        string userInput = Console.ReadLine();

        object key;

        // Attempt to parse user input into different types
        if (int.TryParse(userInput, out int intKey))
        {
            key = intKey;
        }
        else if (double.TryParse(userInput, out double doubleKey))
        {
            key = doubleKey;
        }
        else if (bool.TryParse(userInput, out bool boolKey))
        {
            key = boolKey;
        }
        else if (DateTime.TryParse(userInput, out DateTime dateKey))
        {
            key = dateKey;
        }
        else
        {
            key = userInput; // Assume it's a string if none of the above types
match
        }

        // Retrieve and display the value associated with the entered key
        if (studentGrades.TryGetValue(key, out object value))
        {
            Console.Write($"Key: {key}, Value: ");
```

```
            if (value is List<int> grades)
            {
                Console.WriteLine(string.Join(", ", grades));
            }
            else
            {
                Console.WriteLine(value);
            }
        }
        else
        {
            Console.WriteLine("The entered key does not exist in the
dictionary.");
        }
    }
}
```

18.

Purpose and Benefits of Using Interfaces in C#

Purpose:

- Interfaces define a contract that classes can implement. This contract specifies a set of methods, properties, events, or indexers that the implementing class must provide. However, it does not dictate how these members should be implemented.
- **Abstraction**: Interfaces provide a way to define capabilities without worrying about the specific implementation details, promoting higher levels of abstraction.

Benefits:

1. **Loose Coupling**: By programming to an interface rather than a specific class, you can reduce dependencies between different parts of a program. This makes it easier to change or replace components without affecting the rest of the application. For example, if you have an interface IDrive, you can switch between different implementations (Car, Bike, etc.) without altering the code that uses IDrive.
2. **Code Reusability**: Interfaces allow different classes to implement the same interface, promoting reusability. You can create methods that operate on interface types, making your code more generic and reusable.
3. **Polymorphism**: Interfaces enable polymorphism, allowing objects of different classes that implement the same interface to be treated as objects of the interface type. This allows for more flexible and extensible code.
4. **Testability**: Interfaces make it easier to test code since you can use mock objects that implement the same interface in your unit tests.

18 (a)

```
using System;
using System.Collections.Generic;

// Define the IDrive interface with a Drive() method
public interface IDrive
{
```

```
    void Drive();
}

// Implement the IDrive interface in the Car class
public class Car : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Car is driving.");
    }
}

// Implement the IDrive interface in the Bike class
public class Bike : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Bike is driving.");
    }
}

public class Program
{
    public static void Main()
    {
        // Create a list of IDrive objects
        List<IDrive> vehicles = new List<IDrive>();

        // Add Car and Bike objects to the list
        vehicles.Add(new Car());
        vehicles.Add(new Bike());

        // Demonstrate polymorphism by calling the Drive method on each object
        foreach (IDrive vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }
}
```

18 (b)

Role of Abstract Classes in C# and How They Differ from Interfaces

Role of Abstract Classes:

- **Partial Implementation**: Abstract classes allow you to define a base class that includes both fully implemented methods and abstract methods (methods without implementation). Derived classes must implement the abstract methods.
- **Shared Code**: Abstract classes are useful when you want to provide some shared functionality among a group of related classes while still enforcing certain methods to be implemented by the derived classes.
- **Inheritance**: Abstract classes are designed to be inherited by other classes. They can include fields, constructors, and non-abstract methods, which provide a common base functionality.

Differences Between Abstract Classes and Interfaces:

1. **Method Implementation**:
    - **Abstract Class**: Can contain both abstract methods (without implementation) and non-abstract methods (with implementation).
    - **Interface**: Can only contain method signatures (until C# 8.0, which introduced default interface methods) and no implementation. Starting with C# 8.0, interfaces can have default implementations, but they are not as flexible as those in abstract classes.
2. **Inheritance**:
    - **Abstract Class**: A class can inherit from only one abstract class because C# does not support multiple inheritance.
    - **Interface**: A class can implement multiple interfaces, allowing more flexibility in design.
3. **Fields and Properties**:
    - **Abstract Class**: Can have fields, properties, and constructors, which can be used to maintain state.
    - **Interface**: Cannot have fields or constructors; it only defines members that need to be implemented.
4. **Use Cases**:
    - **Abstract Class**: Use when you need to provide common base functionality and enforce a common contract. It's more appropriate when the derived classes are closely related and share a significant amount of code.
    - **Interface**: Use when you need to define a contract that can be applied to various, potentially unrelated classes. Interfaces are ideal for scenarios requiring loose coupling and multiple inheritance of behavior.

Scenarios Where Abstract Classes May Be More Appropriate:

- **Shared Code**: When multiple classes share a common base of functionality that shouldn't be redefined in each class, an abstract class allows you to define this shared functionality once.
- **Base Class with Default Behavior**: If you have a base class that provides a default behavior that many, but not all, derived classes will use, an abstract class is more appropriate.
- **Controlled Inheritance**: When you want to control and enforce the inheritance structure, ensuring that certain classes can only derive from a specific base class.

18 (c)

```
using System;
using System.Collections.Generic;

// Define the abstract class Animal with an abstract method MakeSound
public abstract class Animal
```

```
{
    public abstract void MakeSound();
}

// Implement the abstract class in the Dog class
public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Dog barks: Woof!");
    }
}

// Implement the abstract class in the Cat class
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Cat meows: Meow!");
    }
}

public class Program
{
    public static void Main()
    {
        // Create a list of Animal objects
        List<Animal> animals = new List<Animal>();

        // Add Dog and Cat objects to the list
        animals.Add(new Dog());
        animals.Add(new Cat());

        // Demonstrate polymorphism by calling the MakeSound method on each
object
        foreach (Animal animal in animals)
        {
            animal.MakeSound();
        }
    }
}
```

19.

Benefits of a Top-Down Approach in Large-Scale Application Development

A top-down approach involves starting with the highest level of the system—typically the overall architecture and design—and breaking it down into smaller, more manageable modules or components. Here's how this approach can benefit a project:

1. **Clear Vision and Structure**:
    - By focusing on the overall structure first, the development team gains a clear understanding of the entire system. This ensures that all components work together seamlessly and align with the project's goals.
    - **Modular Design**: The system is divided into modules that correspond to different functionalities. This modular design makes the system easier to manage, test, and maintain.

2. **Effective Resource Allocation**:
    - **Prioritization**: Key components and functionalities are identified early, allowing the team to allocate resources—time, personnel, budget—more effectively.
    - **Focus on High-Level Requirements**: The top-down approach ensures that high-level requirements and critical components receive attention first, reducing the risk of overlooking important aspects.
3. **Consistency and Standards**:
    - **Unified Design**: By defining the overall architecture first, the project can enforce consistency in design, coding standards, and technology choices across all modules.
    - **Integration Planning**: Since the modules are defined within the context of the whole system, integration between them is smoother, reducing potential conflicts and incompatibilities.
4. **Risk Management**:
    - **Early Identification of Challenges**: High-level design exposes potential risks and challenges early in the project. This allows the team to address these issues proactively.
    - **Better Estimation**: With a well-defined structure, the team can more accurately estimate timelines, costs, and resource needs, minimizing the risk of delays or budget overruns.
5. **Scalability and Flexibility**:
    - **Scalable Design**: The top-down approach enables the design of scalable systems that can handle growth and changes more easily.
    - **Adaptability**: As the project progresses, it's easier to adapt or expand the system because the high-level structure is already in place, guiding modifications.

Example Project Where Top-Down Might Be the Best Approach: An E-Commerce Platform

How a top-down approach could be applied:

1. **Define the Overall Architecture**:
    - **High-Level Components**: Start by identifying the main components: User Management, Product Management, Shopping Cart, Payment Gateway, and Order Management.
    - **Determine Interactions**: Define how these components interact. For instance, the Shopping Cart should interact with Product Management to display products and with the Payment Gateway for processing transactions.

2. **Break Down into Modules**:
    - **User Management**: Break this down into modules like Registration, Login, Profile Management, and Authentication.
    - **Product Management**: Define modules like Product Listings, Categories, Search, and Inventory Management.
3. **Establish Standards and Technologies**:
    - **Choose Technologies**: Decide on the technology stack (e.g., web frameworks, databases, payment gateways) to be used consistently across the platform.
    - **Set Standards**: Define coding standards, security practices, and API protocols for seamless integration between modules.
4. **Detailed Design of Each Module**:
    - **User Interface**: Design the user interface for each component, ensuring a consistent look and feel across the platform.
    - **Database Design**: Create a database schema that supports all modules and ensures data integrity and performance.
5. **Implementation and Testing**:
    - **Develop Modules**: Each module is implemented according to the high-level design. For instance, the Payment Gateway module would be developed to handle various payment methods.
    - **Integration Testing**: Once individual modules are implemented, integration testing ensures that they work together as planned.
6. **Deployment and Maintenance**:
    - **Scalability**: The platform is designed to scale with increased traffic, new product categories, or additional features like international shipping.
    - **Ongoing Updates**: As the platform evolves, the top-down structure allows for easier implementation of new features or technologies.

19 (b)

Benefits of a Bottom-Up Approach in Application Development

A bottom-up approach begins with the development of small, independent functions or components. These smaller units are thoroughly tested and refined before being combined into larger modules and, eventually, the complete system. Here's how this approach can lead to a more flexible and testable application:

1. **Focus on Reusability**:
    - **Building Reusable Components**: Starting with small, independent functions encourages the creation of reusable components. These components can be tested in isolation, ensuring they work correctly before integration.

- **Modularity**: Each function or module is designed to perform a specific task. These modular components can be easily reused in different parts of the application or even in other projects, enhancing code reusability.

2. **Ease of Testing**:
    - **Unit Testing**: Small functions can be thoroughly unit-tested, making it easier to identify and fix bugs early in the development process. This leads to higher-quality code.
    - **Test-Driven Development (TDD)**: The bottom-up approach aligns well with TDD practices, where tests are written first for small units of functionality, guiding the development process.
3. **Flexibility and Adaptability**:
    - **Incremental Development**: By developing and testing small components first, the project can adapt to changes more easily. If a requirement changes, it often affects only a few small components rather than the entire system.
    - **Easier Refactoring**: Small, well-defined components are easier to refactor or replace without affecting the rest of the system, making the application more adaptable to future changes.
4. **Scalability and Integration**:
    - **Scalable Solutions**: The bottom-up approach allows for the gradual scaling of the application. As the system grows, new components can be added and integrated with minimal disruption.
    - **Seamless Integration**: Since each component is independently tested, integrating them into larger modules is usually smoother. The application grows organically, with each piece fitting into the larger system.
5. **Parallel Development**:
    - **Team Collaboration**: Different team members can work on different components simultaneously, speeding up development. Because the components are independent, teams can develop and test them in parallel before integration.

Example Project Where Bottom-Up Might Be the Best Approach: A Microservices-Based Application

Microservices-based application, such as a financial services platform handles various tasks like account management, transaction processing, reporting, and customer notifications. Here's how a bottom-up approach could be applied.

1. **Develop Independent Microservices**:
    - **Account Management Microservice**: Start by developing a microservice that handles account creation, updates, and deletion. This microservice would

include functions for validating account details, storing them in a database, and retrieving them when needed.

- **Transaction Processing Microservice**: Create another microservice responsible for processing transactions. It would include functions for validating transactions, applying business rules, and updating account balances.

2. **Thorough Testing of Microservices**:
   - **Unit Testing**: Each microservice is thoroughly unit-tested to ensure that its individual components work correctly. For example, the transaction validation function is tested to handle various scenarios (e.g., insufficient funds, currency conversion).
   - **Mock Testing**: Using mock objects, you can test the microservices in isolation, simulating interactions with external services (e.g., payment gateways) without needing the entire system in place.
3. **Combine Microservices into Larger Units**:
   - **Integration of Account and Transaction Services**: After the microservices are tested independently, they are integrated to allow seamless interaction. For instance, when a transaction is processed, it updates the relevant account in the Account Management Microservice.
   - **Building Additional Services**: Additional microservices, such as Reporting and Notification Services, are developed and integrated gradually. Each new service builds on the functionality of the existing services.
4. **Implement and Test Larger Modules**:
   - **End-to-End Testing**: Once the microservices are integrated, end-to-end testing is performed to ensure the entire transaction flow (from account update to transaction processing to notifications) works as expected.
   - **Flexibility in Scaling**: If the platform needs to support new features, such as loan management or investment tracking, new microservices can be developed and integrated without disrupting the existing system.
5. **Deployment and Maintenance**:
   - **Independent Deployment**: Each microservice can be deployed independently, allowing for continuous integration and deployment. If an issue arises in one service, it can be fixed and redeployed without affecting the entire system.
   - **Scalability**: The platform can easily scale by deploying more instances of individual microservices based on demand, ensuring high availability and performance.

https://github.com/Mkweha/C-Assignment/blob/main/C%23_Assignment.pdf

https://github.com/Mkweha/C-Assignment